Composition of Web Services based on Timed Mediation.

Nawal Guermouche, Claude Godart

Composition of Web Services based on Timed Mediation

Nawal Guermouche[1,2]
[1]CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2]Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France
nawal.guermouche@laas.fr
and
Claude Godart
LORIA-INRIA-UMR 7503
F-54506 Vandoeuvre-les-Nancy, France
claude.godart@loria.fr

In the last few years there has been an increasing interest for Web service composition which is one of the important aspects of the Service Oriented Computing (SOC) paradigm. In the literature, many frameworks investigated mechanisms for synthesizing a composition based on operations and/or messages choreography constraints. Apart from these constraints, Web services interactions depend also on crucial *quantitative timed properties*. These properties affect considerably the behavior of services and we need to consider them when synthesizing compositions. Since Web services are developed autonomously, when composing services, conflicts, and in particular timed conflicts can arise and the composition fails. An interesting alternative is to generate a mediator to try to avoid these timed (and non timed) conflicts. In this paper, we first present how to model the behavior of Web services augmented with timed properties, and second we present a mechanism to deal with timed properties when building a composition of asynchronous services, potentially, based on a timed mediator.

Keywords: SOA, Web services composition, asynchronous Web service, timed properties, timed automata, timed mediation

1. INTRODUCTION

With the proliferation of e-business technologies, service oriented computing is increasingly gaining acceptance as one of the promising paradigms in order to achieve cross-organizational interoperability and collaboration. Consequently, enterprises are adopting Web services to outsource their internal business processes and make them accessible via the Web. One of the main features of Web services is the *composition* aspect that allows to dynamically combine services to provide new value added services.

In real life scenarios, Web service aspects, and more particularly Web service composition depends not only on message choreography constraints, but also on other properties such as privacy [18], security [25] issues and *timed properties* [3; 15; 30]. Particularly, in this paper we are interested in timed properties which specify the *necessary delays* to exchange messages (e.g., in an e-government application a prefecture must send its final decision to grant an handicapped pension to a requester after 7 days and within 14 days). Building a correct composition requires to consider message choreography constraints augmented with timed properties. Few recent

works have shown the importance to deal with timed properties in the compatibility analysis of synchronous [33], asynchronous Web services [17; 14], in checking requirements satisfaction [21], and in calculating temporal thresholds for process activities [30].

Due to the autonomous nature of services development, interfaces and protocols mismatches can arise. In that case, the composition fails. This happens when there are services that wait for messages that cannot be received, i.e., when: (1) the awaited messages are not produced by other services, (2) the awaited and sent messages are not adequate (i.e., the awaited and the sent messages have different names or different data types), (3) there is a mutual services blocking (e.g., a service Q_1 waits for a message m that must be sent by another service Q_2, which also waits for a message m' from Q_1 to send the awaited message m), and (4) timed properties of the sent and the awaited message are conflicting. In this paper, we focus particularly on time related issues. The composition of timed services is a difficult problem [15]. Indeed, when composing services, timed dependencies can be built and can give rise to implicit timed conflicts and the composition fails. In this context, it is mandatory to define mechanisms to discover theses implicit timed conflicts and to try to avoid them in order to fulfil a composition.

One of important ingredients we need in a timed composition framework is the Web services *timed behavior description*. The timed behavior of a Web service specifies supported sequences of messages, involved data types, constraints over data, and associated timed requirements. Such description is called *timed conversational protocol* (for short we say conversational protocol). To specify such conversational protocols, we use a finite state machine (FSM) based model. This kind of formal representation has been already used in a series of work [10; 33; 27; 16] and seems adequate. In fact, a state machine based model is suitable to describe reactive behaviors [4], it is fairly easy to understand, and at the same time it is expressive enough to model the properties we consider. More specifically, we rely on clocks as defined in standard timed automata to model timed properties [1].

To summarize, the problem we are interested in can be defined as follows: given a timed description of given need, called a *client service*, and a set of discovered timed services, how to build a composition of the set of discovered services to satisfy the client service. We note that we focus on correct interactions of services and we do not consider exception handling which are out of scope of this paper. The main contributions of our framework are as follows:

(1) Unlike the existing composition synthesis models, we propose a formal model of asynchronous Web services that takes into account *timed properties* associated to messages, data, and data constraints.

(2) As we deal with timed properties, when synthesizing a composition, implicit timed conflicts can arise. We propose a mechanism to discover timed conflicts when building a composition.

(3) In addition, we propose the use of a mediator based process to solve, when possible, the problem of timed (and non timed) conflicts when building a composition.

(4) Finally, the primitives described in this paper have been implemented in a prototype that we used to perform preliminary tests.

The reminder of the paper is organized as follows: in Section 2 we present a global overview of our framework. Section 3 describes how we model a timed conversational protocol as a finite state machine specification equipped with clocks. Section 4 describes our composition approach steps. Section 5 presents a concrete example of composition to illustrate our approach. A validation setup is presented in Section 6. Related work is introduced in Section 8, and finally Section 9 concludes.

2. GLOBAL OVERVIEW

In this section, we present an overview of our timed composition framework which relies on the following elements:

A client Service: the first element of our framework is the timed description of the client service. This service specifies timed properties associated to the data flow the client provides and to the data flow he expects without any reference to the operations of available services.

A set of discovered services: we assume that a set of timed Web services can be discovered to answer the client service request.

A mediator: it can access the data yet exchanged by the different services and use them to generate any missing messages.

Case study: e-government application

Let us present a part of an e-government application inspired from [24] to illustrate the related issues of the problem we handle. The goal of the e-government application we consider is to manage handicapped pension requests. Such a request involves three organizations: (1) *a prefecture*, (2) *a health authority*, and (3) *a town hall*. We suppose that theses organizations are managed by, respectively, the prefecture service (PS), the health authority service (HAS), and the town hall service (THS).

A high level choreography model of the process is depicted in Fig. 1. A citizen can apply for a pension. Once applied, the prefecture solicits the medical entity to deliver an examination report of the requester, and the town hall to deliver the domiciliation attestation. After studying the received files, the prefecture sends the notification of the final decision to the citizen. The interaction between these partners is constrained by timed requirements:

Once the health authority service proposes meeting dates to the citizen, this one must confirm the meeting within 24 hours.

The prefecture requires at least 48 hours and at most 96 hours from receiving the file from the requester to notifying the citizen with the final decision.

The medical report must be sent to the prefecture after at least 120 hours and at most 168 hours after receiving the request of the medical report.

Notion of timed conflicts

Given this set of timed Web services and the client service, our aim is to build a timed composition that satisfies this client service. When building a composition, it is mandatory to ensure that data and timed constraints of the involved services are not conflicting. In the context of our work, we do not focus on data type and semantics related analysis problems. We consider simple data which can be simply checked: two data constraints are said to be not conflicting if their solution set is not disjoint. For example, the prefecture studies the pension request only if the requester is at least 16 years old. If we want to create a connection between the requester and the prefecture service to exchange the pension request while the requester is for example at least 18 old, this is possible (i.e., the set of solution of age$\geq 16 \cap$ the set of solution of age$\geq 18 \neq \emptyset$).

While the data constraints we consider can be checked by verifying their set of solutions, timed constraints validation needs more complex investigations. In fact, in a collaboration, timed properties of Web services cannot be checked like simple constraints. In other words, to assert that an interaction is timed deadlock free, it is not sufficient to check timed constraints assigned to sending a message with timed constraints associated to its reception. For example, the prefecture must send its final decision after 48 hours and within 96 hours from receiving the pension request. On the other side, the requester must receive it within 120 hours from sending the request. If we check these two timed constraints as simple constraints, we can conclude that the prefecture and the requester can collaborate together. However, if we examine the progress

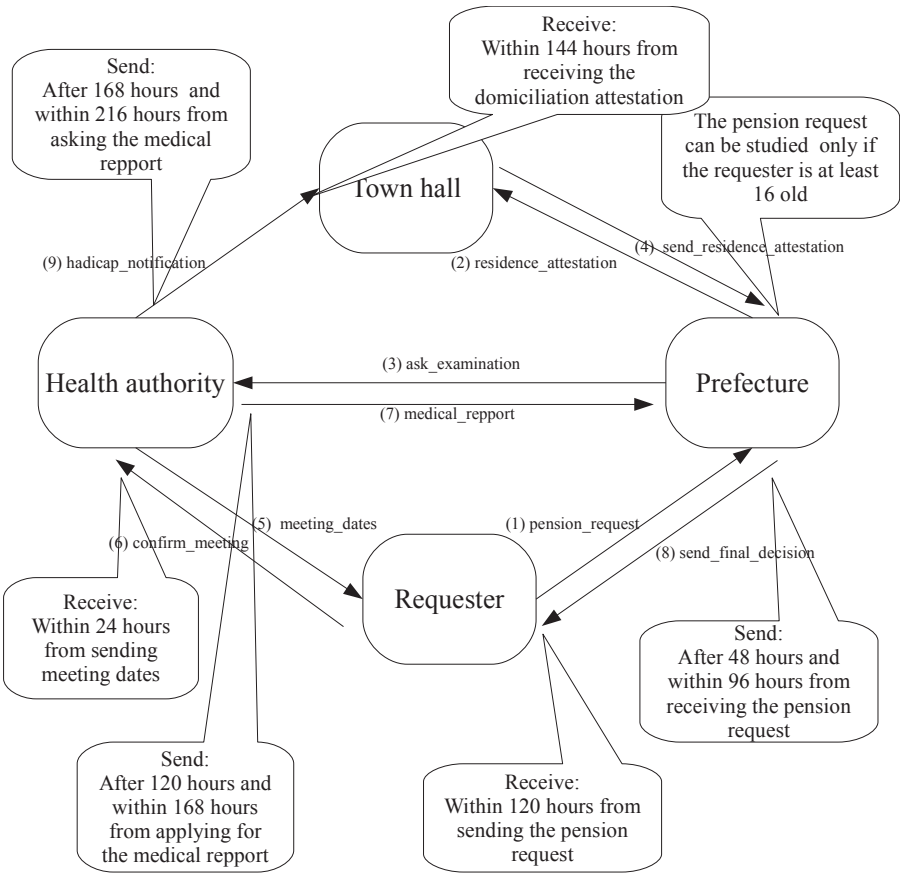


Figure. 1. Global view of the e-government application

of the interaction, we can remark that the prefecture can send its final decision only after the medical report has been received. This report must be sent by the medical entity after 120 hours and within 168 hours from receiving the report request. Since the prefecture must wait for the medical report to send its final decision, i.e., after 120 hours, the final decision cannot be sent within 96 hours from receiving the pension request. Fig. 2, illustrates this conflicting interaction. The prefecture sends its decision after 48 hours and within 96 hours from receiving the pension request. But during this execution, the prefecture must wait for at least 120 hours to get the medical report. This presents a simple timed conflict. More complex timed conflicts can arise and can make fail the composition. As said previously, to succeed the composition, an alternative consists in generating a mediator whose role is to try to prevent these conflicts.

Now, let us check the scenario depicted in Fig. 1. We can remark that the town hall has to wait for a medical report of the medical entity before to, for example, deliver an handicapped card. The town hall must receive the report within 144 hours, but the medical entity can send its report only after 168 hours. So, the town hall cannot receive the report in time and the composition will fail.

But, if we examine the situation in details, we can remark that the medical entity sends its report to the prefecture after 120 hours. As a consequence, intuitively, to succeed the collaboration, we can build an indirect connection between the medical entity and the town hall to deliver the medical report within 144 hours. This indirect connection can be built by the mediator that generates the message for transmitting the medical report to the town hall in advance. Note that

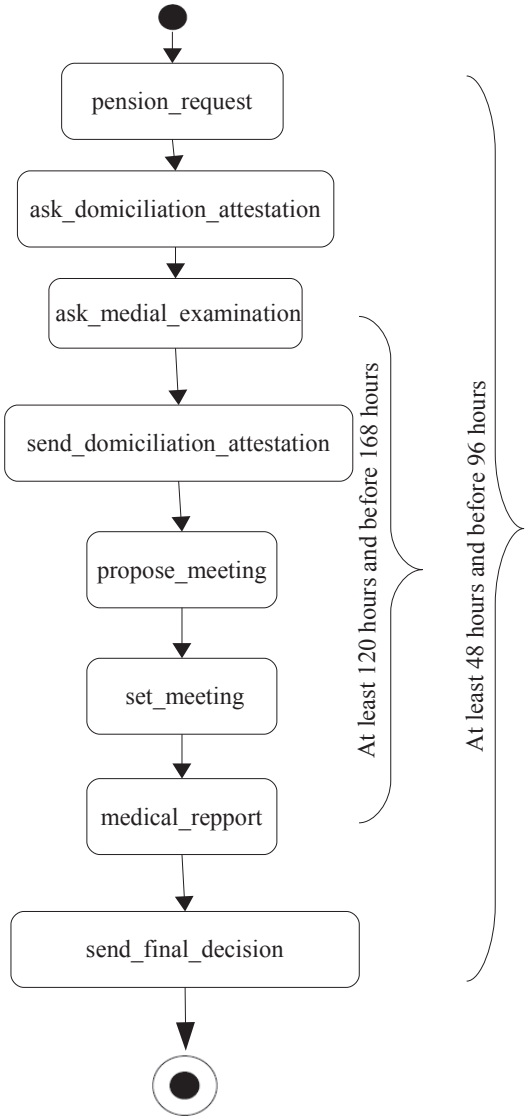


Figure. 2. Example of impact of timed properties on Web services interaction

the mediator fails when a required data (i.e., the data involved in the required message) is not available (i.e., the data is not accessible).

To summarize, in this section we have intuitively discussed the impact and the importance to consider timed properties in a composition framework. During a composition, different services with different timed constraints can be involved. Timed properties can give rise to timed conflicts which can make fail the composition. In the following, we propose a formal approach which aims at composing services so that their timed properties are respected.

3. MODELING THE TIMED BEHAVIOR OF WEB SERVICES

One of the important ingredient in a composition framework is the *timed conversational protocol* of Web services which we assume deterministic. In our framework, the timed conversational protocol specifies the sequences of messages a service supports, the involved data flow and

the associated timed properties to exchange messages. We adopt deterministic timed automata based formalism to model the timed behavior of Web services (i.e., the timed conversational protocol). Intuitively, the states represent the different phases a service may go through during its interaction. Transitions enable sending or receiving a message. An output message is denoted by $!m$, whilst an input one is denoted by $?m$. A message involving a list of data is denoted by $m(d_1, \ldots, d_n)$, or $m(\bar{d})$ for short. To capture the timed properties when modelling Web services, we use standard timed automata clocks [1]. The automata are equipped with a set of clocks. The values of these clocks increase with the passing of time. Transitions are labelled by timed constraints, called *guards*, and resets of clocks. The former represent simple conditions over clocks, and the latter are used to reset values of certain clocks to zero. The guards specify that a transition can be fired if the corresponding guards are satisfiable.

Let X be a set of clocks. The set of *constraints* over X, denoted $\Psi(X)$, is defined as follows:
true $\mid x \bowtie c \mid \psi_1 \wedge \psi_2$, where $\bowtie \in \{\leq, <, =, \neq, >, \geq\}$, $x \in X$, $\psi_1, \psi_2 \in \Psi(X)$, and c is a constant.

Definition 1. *A* timed conversational protocol *Q is a tuple (S, s_0, F, M, C, X, T) where S is a set of states, s_0 is the initial state, F is a set of final states ($F \subseteq S$), M is a set of messages, C is a set of constraints over data, X is a set of clocks, and T is a set of transitions such that $T \subseteq S \times M \times C \times \Psi(X) \times 2^X \times S$ with an element of the alphabet (exchanged message (M)), a constraint over data (C), a guard over clocks ($\Psi(X)$), and the clocks to be reset (2^X).*

The conversational protocols we consider are deterministic. A conversational protocol is said to be deterministic if for each two transitions $(s, \alpha_1, c_1, \psi_1, s'_1)$ and $(s, \alpha_2, c_2, \psi_2, s'_2)$, the following conditions are satisfied:

$\alpha_1 \neq \alpha_2$, or
$c_1 \wedge c_2 = \text{false}$, or
$\psi_1 \wedge \psi_2 = \text{false}$

Example 1. *Fig. 3 illustrates the timed conversational protocol of the PS, THS, HAS services of the use case study, and the client service. In this figure, the initial state of the PS service is p_0, the set of states is $\{p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}, p_{16}\}$ and the set of final states is $\{p_7, p_9, p_{16}\}$. This service can send and receive messages. For example, it can send the message examination_request(sn, handicap), denoted !examination_request(sn, handicap). This message has as parameters the security number (sn), and the handicap (handicap) of the requester. Analogously, this service can consume a message, for example, the message pension_request(sn, age, handicap), denoted ?pension_request(sn, age, handicap). This message has as parameters the security number (sn), the age (age), and the handicap (handicap) of the requester. This service achieves correctly its execution if for each interaction it reaches a final state.*

To specify that the prefecture must send its final decision with a delay of 48 to 96 hours after receiving the pension request, we associate to the reception of the request of the pension a reset of a clock t_1 ($t_1 := 0$) and we assign the constraint $48 \leq t_1 \leq 96$ to the sending of the final decision.

4. ANALYZING THE TIMED COMPOSITION PROBLEM

In this section, we present the algorithm that allows to synthesize a composition of timed Web services. We recall that our framework gathers three steps: (1) creating timed P2P connections between the client service and the discovered services (see Section 4.1), (2) discovering timed conflicts (see Section 4.2), (3) and generating a *mediator* that tries to step in to succeed a connection (see Section 4.3).

4.1 Building timed P2P connections

Given a set of conversational protocols of the services and the client service, our aim is to build a *timed global automaton* that characterizes the *timed composition schema* (the global automaton is called *Timed Composition Schema Automaton TCSA*).

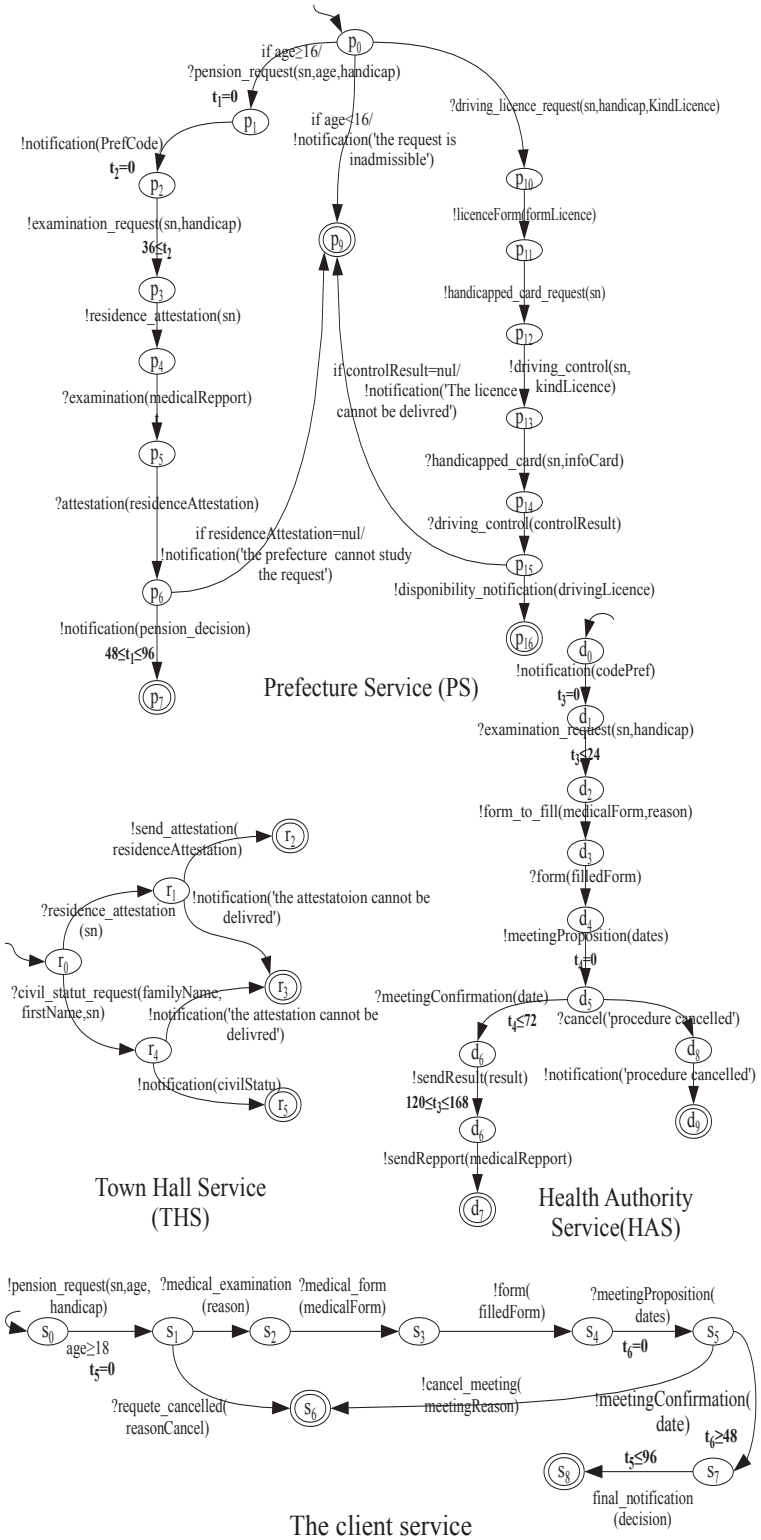


Figure. 3. Services of the e-government scenario

To build this TCSA, we introduce the concept of *configuration* that represents the states of the TCSA at a given time. A configuration defines the evolution of services states when they are interacting together (i.e., connected via channels). In the initial configuration, all the services are in their initial states. Given a source configuration, the TCSA reaches a new configuration when there exists two services that change their states by exchanging a message so that no timed conflict arises.

Definition 2. *(A Timed Composition Schema Automaton)*
A timed composition schema automaton TCSA is a tuple (S, Q, M, X, L, T) *such that S is a set of configurations, Q is a set of services, M is a set of messages, L is a set of channels, X is a set of clocks, and T is a set of TCSA transitions such that* $T \subseteq S \times L \times \Psi(X) \times S$. *A transition specifies that, from a source configuration, the TCSA reaches a new configuration when a channel can be created to interconnect two services so that the associated (ordered) timed constraints are satisfied. The set of channels L is defined as a set of* $(p_s, p_r, m(\bar{d}))$, *with* $p_s, p_r \in Q$, *and the tuple* $(p_s, p_r, m(\bar{d}))$ *specifies that the service* p_s *sends the message* $m(\bar{d})$, *that involves the set of data types* $(\bar{d})$, *to the service* p_r. *In our composition framework, a mediator can be generated, hence the set of considered services is* $Q = \{R, A, Med\}$, *such that R is the client service, A is the set of the available services, and Med is the generated mediator.*

Among the transitions of the different services, we distinguish two kinds of transitions: *passive transitions* and *non-passive transitions*.

A *passive transition*: is a timed (resp. non-timed) transition that has timed constraints of the form $x \leq v$ (resp. $x < v$). In fact, these transitions are considered passive because they do not give rise to timed conflicts.

A *non-passive transition*: is a timed transition that has timed constraints of the form $x \geq v$ (resp. $x > v$). In fact, timed conflicts can arise when these transitions precede transitions that have constraints of the form $x \leq v$ (resp. $x < v$).

The approach of composition is based on the algorithm 4.1. This algorithm aims to build connections between the different services to try to satisfy the client service. The steps of this algorithm can be described as follows:

From the set of transitions T, it isolates passive transitions T_p and non-passive transitions T_{np}. Initially, it tries to connect each transition of the client service with the transitions of the different services. Note that this algorithm tries to connect passive transitions before non-passive transitions. In fact, the study we have performed shows that timed conflicts can arise when non-passive transitions precede passive transitions. When the connection fails, this algorithm calls the algorithm 4.3 that aims at generating the mediator. When a connection is created, the algorithm 4.2 checks if the created connection does not give rise to timed conflict. In Section 4.2, we present the process of discovering timed conflicts.

The steps that we are using to build a *TCSA*, numbered according to the (polynomial) algorithm 4.1 are described bellow:

(1) For each transition of each trace of the client service,
(2) From each current state of the Web services, we isolate the passive transitions,
(3) Similarly, we isolate the non-passive transitions,
(4) If the current transition of the client service is a passive transition and if it can be connected with a transition of Web services (see Algorithm 4.2), then
(5) We update the current state of the client service,
(6) Else, if the connexion fails, then
(7) We check if there is a transition that can be connected among the passive transitions of the Web services,
(8) If there is no a passive transition that can be connected, then

Algorithm 4.1: Composition

Input: A client service $Q_g = (S_c, s_{0_c}, F_c, M_c, C_c, X_c, T_c)$, a set of Web services $Q_i = (S_i, s_{0_i}, F_i, M_i, C_i, X_i, T_i)$, for $i = \{1, .., n\}$, the initial configuration of the TCSA $\overline{s} = (s_{0_1}, \ldots, s_{0_n})$, the current state of the client service $s_c = s_{0_c}$

Output: TCSA $= (S, s_0, F, M, C, X, T)$, and the mediator $Med = (S_{med}, s_{0_{med}}, F_{med}, M_{med}, C_{med}, X_{med}, T_{med})$

begin

succesComposition=true;

1 **for** *each transition* $t_c = (s_c, \alpha_c, c_c, \psi_c, Y_c, s'_c)$ *of each trace of the client service and* ***if*** *succesComposition* **do**

2 $T_{sp} = \{t_i\}$, for $i = \{1..n\}$ where $t_i = (s_i, \alpha_i, c_i, \psi_i, Y_i, s_{i+1})$ such as $s_i \in (s_1 \ldots s_n)$ and $\psi_i = x \leq c$ or $\psi_i = x < c$;

3 $T_{snp} = \{t_j\}$, for $j = \{1..n\}$ where $t_j = (s_j, \alpha_j, c_j, \psi_j, Y_j, s_{j+1})$ such as $s_j \in (s_1 \ldots s_n)$ and $\psi_j = x \geq c$ or $\psi_j = x > c$;

$T_{visited} = null$;

4 **if** *((*$\psi_c = x < c$*) or (*$\psi_c = x \leq c$*)) and (satisfaction(*t_c*,*$\overline{s}$*))* **then**

5 $s_c = s'_c$;

else

6 **if** *((*$\psi_c = x < c$*) or (*$\psi_c = x \leq c$*)) and (*$\neg$*satisfaction(*t_c*,*$\overline{s}$*))* **then**

choose onf transition t_{sp} of T_{sp};

$T_{visited} = T_{visited} \cup t_{sp}$;

7 **while** $T_{sp} \nsubseteq T_{visited}$ *and* $\neg$ *satisfaction(*$t_{sp}, \overline{s}$*)* **do**

choose another transition t_{sp} of T_{sp};

$T_{visited} = T_{visited} \cup t_{sp}$;

8 **if** $\neg$ *satisfaction(*t_{sp}*,*$\overline{s}$*)* **then**

Choose one transition t_{snp} de T_{snp};

$T_{visited} = T_{visited} \cup t_{snp}$;

9 **while** $T_{snp} \nsubseteq T_{visited}$ *and* $\neg$ *satisfaction(*$T_{snp}, \overline{s}$*)* **do**

choose another transition t_{snp} of T_{snp};

$T_{visited} = T_{visited} \cup t_{snp}$;

10 **if** $\neg$ *satisfaction(*t_{snp}*,*$\overline{s}$*)* **then**

11 succesComposition=false;

else

12 **if** *(*$\psi_c = x > v$*)or(*$\psi_c \geq v$*)* **then**

choose a transition t_{sp} of T_{sp};

$T_{visited} = T_{visited} \cup t_{sp}$;

13 **while** $T_{sp} \neq \emptyset$ *and* $\neg satisfaction(T_{sp}, \overline{s})$ **do**

choose another transition t_{sp} of T_{sp};

$T_{visited} = T_{visited} \cup t_{sp}$;

14 **if** $\neg$ *satisfaction(*t_{sp}*,*$\overline{s}$*)* **then**

15 **if** *satisfaction(*$t_c, \overline{s}$*)* **then**

16 $s_c := s'_c$

else

17 choose a transition t_{snp} of T_{snp};

$T_{visited} = T_{visited} \cup t_{snp}$;

18 **while** $T_{snp} \neq \emptyset$ *and* $\neg satisfaction(T_{snp}, \overline{s})$ **do**

choose another transition t_{snp} of T_{snp};

$T_{visited} = T_{visited} \cup t_{snp}$;

19 **if** $\neg$ *satisfaction(*t_{snp}*,*$\overline{s}$*)* **then**

20 succesComposition=false;

21 **if** *succesComposition* **then**

22 **if** $\overline{s} \in F$ **then**

23 Return ASCT and the mediator;

else

24 'The composition fails because the services do not reach their final states';

else

'The composition fails because the client service cannot be satisfied';

(9) We check if there is a transition that can be connected among the set of non-passive transitions of the set of Web services,
(10) If there is no a transition that can be connected, then
(11) The composition fails
(12) If the transition of the client service is non-passive, then
(13) Before trying to connect the current transition of the client service, we verify if there is a passive transition of the services that can be connected,
(14) If no transitions can be connected, then
(15) If the non-passive transition of the client service can be connected, then
(16) We update the current state of the client service,
(17) Else, if the connection of the non-passive transition of the client service fails, then
(18) We try to connect one non-passive transition of the services,
(19) If the connection fails,
(20) Then, the composition fails,
(21) If the composition did not fail, then,
(22) If the services reach their final states, then
(23) Return the TCSA and the generated mediator,
(24) Else, the composition fails

The function *satisfaction()* (see the algorithm 4.2) is used to create P2P connections (i.e., channels) between two services. This function has two inputs, a transition $(s, \alpha(\bar{d}), c, \psi, Y, s')$ to be connected and the current configuration of the services $\bar{s}$. The related steps, which are numbered according to the algorithm 4.2, are presented below.

(1) First, we check if the transition $(s, \alpha(\bar{d}), c, \psi, Y, s')$ is not already visited, then
(2) We check if there is a service that enables a passive transition $(s_j, \alpha_j(\bar{d}), c_j, \psi_j, Y_j, s'_j)$ such that: the constraints over data c and c_j are not-disjoint, and the messages $\alpha(\bar{d})$ and $\alpha_j(\bar{d})$ are equal but with a different polarity[1]. That means, if the transition $(s, \alpha(\bar{d}), c, \psi, Y, s')$ enables an input message (resp. output message), the transition $(s_j, \alpha_j(\bar{d}), c_j, \psi_j, Y_j, s'_j)$ must enable the output counterpart (resp. the input counterpart).
(3) In our framework, the mediator can be generated to consume extra messages. These messages can be sent within a timed interval. In this case, the timed constraints associated to these messages will be associated to the mediator transitions. Thus, for each reset of clocks Y and Y_j, we generate an empty mediator that enables to reset the same clocks (Y, Y_j). In fact, these clocks can be used later by the mediator to specify the timed constraints associated to consuming extras messages.
(4) If an empty mediator is generated $(s_{med}, \epsilon, Y, Y_j, s'_{med})$ to reset clocks, then we build a TCSA that connects the two transitions $(s, \alpha(\bar{d}), c, \psi, Y, s')$ and $(s_j, \alpha_j(\bar{d}), c_j, \psi_j, Y_j, s'_j)$ with the generated empty transition of the mediator $(s_{med}, \epsilon, Y, Y_j, s'_{med})$.
(5) Else, if there is no clocks reset, the mediator remains in the same state. We build a TCSA that connects the two transitions $(s, \alpha(\bar{d}), c, \psi, Y, s')$ and $(s_j, \alpha_j(\bar{d}), c_j, \psi_j, Y_j, s'_j)$.
(6) We apply the ordering process to detect the possible timed conflicts (via the function *clock_order*() presented by the algorithm 4.3). If the transition is deadlock-free, then
(7) The candidate TCSA transition is accepted,

[1]The polarity of a message defines if the message is an input or an output one. $Polarity(m) =?$ if m is an input message. $Polarity(m) =!$ if m is an output message

(8) We update the current configuration of the Web services and return true
(9) Else, if a conflict arises, we call the mediator (see the algorithm 4.4)
(10) If a mediator is generated, then we generate a candidate TCSA transition that connects the current transition with the transition of the mediator,
(11) We apply the clock ordering process. If there is no conflict, then
(12) We accept the candidate transition,
(13) We update the current configuration of the services and return true
(14) If the connexion of the transition $(s_i, \alpha_i(\overline{d}), c_i, \psi_i, Y_i, s_i)$ fails, then we apply the above steps by using the non-passive transitions of the services.
(15) Applying the above steps, if the connexion fails, then, return false.

In the following, we present the process of discovering timed conflicts.

4.2 Making explicit the implicit timed constraints dependencies

As said previously, when creating TCSA transitions, implicit timed dependencies can be created. In that case, timed conflicts can arise. In order to discover timed conflicts when combining services, we need a mechanism that allows to make explicit the implicit timed dependencies. To do so, we propose the *clock ordering* process. The idea behind the *clock ordering* process is to define an order between the different clocks of the services for each new TCSA transition.

To explain why simple checking of timed constraints as simple constraints (called local checking) is not sufficient to ensure a correct composition, we consider the following example depicted in Fig. 4.

Example 2. *Let us consider the two timed conversational protocols P and P′. We start by building the TCSA of the two conversational protocols by considering the timed constraints as simple constraints over data, i.e., we check locally the timed constraints of the transitions.*

As we can see, the service P sends the message m_0 and resets a clock x. The service P′ can receive this message. So we can build the TCSA transition $(s_0s'_0, m_0, x_1 = 0, s_1s'_1)$. Then the service P′ sends the message m_1 and resets the clock y. The service P can receive the message m_1. We build the TCSA transition $(s_1s'_1, m_1, y = 0, s_2s'_2)$. Later, the service P sends the message m_2, the service P′ can receive it after 20 units of time. Hence, we build the TCSA transition $(s_2s'_2, m_2, y \geq 20, s_3s'_3)$. After that, the service P′ sends the message m_3, the service P can receive it within 10 units of time. We build the TCSA transition $(s_3s'_3, m_3, x < 10, s_4s'_4)$. As we can see in Fig. 4.(a), by simply checking timed constraints of transitions, we could build a TCSA.

However, the message m_2 can be exchanged after 20 units of time and the message m_3 can be exchanged within 10 units of time. As m_3 can be exchanged after exchanging the message m_2, hence it can be exchanged after 20 units of time, i.e., the message m_3 can be exchanged within 10 (i.e., [0,10]) units of time and after 20 (i.e., [20,∞)) units of time. This latter represents a timed conflict. To cater for such implicit timed properties, we propose to perform a clock ordering process. This process allows to define an order between the clocks of the TCSA transitions. Below, we show how we define the clock order.

The two services can exchange the message m_0 via the TCSA transition $(s_0s'_0, m_0, x = 0, s_1s'_1)$. Then when building the TCSA transition $(s_1s'_1, m_1, y = 0, s_2s'_2)$ we can define the order $y \leq x$ since y is reset after x. So we associate this order to the TCSA transition as follows $(s_1s'_1, m_1, 0 \leq y \leq x, s_2s'_2)$. Then, the service P can send the message m_2 to the service P′ which can receive after 20 units of time. So when the two services exchange the message m_2, $(0 \leq y \leq x) \wedge (y \geq 20)$ must be satisfied. We build the TCSA transition $(s_2s'_2, m_2, 0 \leq y \leq x, y \geq 20, s_3s'_3)$. Until now, there is no timed conflict. Note that we propagate the constraint $y \geq 20$ over the successor transitions. When the service P′ sends the message m_3, the service P can receive it within 10 units of time, i.e., $20 \leq y \leq x \leq 10$ must be satisfied. However, this latter induces to a timed conflict ($20 \leq 10$). As we can see in Fig. 4.(b), by defining a clock ordering when combining services, implicit timed conflicts can be discovered.

Algorithm 4.2: Satisfaction

Input: A transition $t_i = (s_i, \alpha_i(\overline{d}), c_i, \psi_i, Y_i, s'_i)$, a configuration $\overline{s}$
Output: boolean
P2PConnection($(s_i, \alpha_i(\overline{d}), c_i, \psi_i, Y_i, s'_i)$***,*** $\overline{s}$***)***;
begin
  falure=false;
1 **if** $\neg cycle(s_i, \alpha_i, c_i, \psi_i, Y_i, s'_i)$ **then**
2 **if** *there exists* $t_j = (s_j, \alpha_j, c_j, \psi_j, Y_j, s'_j) \in T_{sp}$ *such as* $\alpha_i(\overline{d}) = \overline{\alpha_j(\overline{d})}$ *and* $c_i \cap c_j \neq \emptyset$ **then**
3 **if** $Y_i \neq \emptyset$ *or* $Y_j \neq \emptyset$ **then**
$Y_{asct} = Y_i \cup Y_j$;
$T_{med} = T_{med} \cup (s_m, \epsilon, Y_{asct}, s'_m)$;
$\psi_{asct} = \psi_i \cup \psi_j$;
4 $t_{asct} = (s_0 s_1 \dots s_i \dots s_j \dots s_n s_m, \alpha(\overline{d}), \psi_{asct}, Y_{asct}, s_0 s_1 \dots s'_i \dots s'_j s_n s'_m)$;
else
5 $t_{asct} = (s_0 s_1 \dots s_i \dots s_j \dots s_n s_m, \alpha(\overline{d}), \psi_i, \psi_j, Y_i, Y_j, s_0 s_1 \dots s'_i \dots s'_j s_n s_m)$;
6 **if** $clock\_order(t_{asct})$ **then**
7 $T_{asct} = T_{asct} \cup t_{asct}$;
8 $\overline{s} = s_0 s_1 \dots s'_i \dots s'_j s_n s'_m$, return true;
else
9 $t_{med} = mediateur(t_i, D)$;
if $t_{med} \neq null$ **then**
10 $t_{asct} = (s_0 s_1 \dots s_i \dots s_n s_m, \alpha(\overline{d}), \psi_i, Y_i, s_0 s_1 \dots s'_i \dots s_n s'_m)$;
11 **if** $clock\_order(t_{asct})$ **then**
12 $T_{asct} = T_{asct} \cup t_{asct}$;
13 $\overline{s} = s_0 s_1 \dots s'_i \dots s_n s'_m$, return true;
else
failure= true;
else
failure=true;
else
failure=true
14 **if** *failure* **then**
if *there exists* $t_j = (s_j, \alpha_j, c_j, \psi_j, Y_j, s'_j) \in T_{snp}$ *such as* $\alpha_i(\overline{d}) = \overline{\alpha_j(\overline{d})}$ **then**
if $Y_i \neq \emptyset$ *ou* $Y_j \neq \emptyset$ **then**
$Y_{asct} = Y_i \cup Y_j$;
$T_{med} = T_{med} \cup (s_m, \epsilon, Y_{asct}, s'_m)$;
$\psi_{asct} = \psi_i \cup \psi_j$;
$t_{asct} = (s_0 s_1 \dots s_i \dots s_j \dots s_n s_m, \alpha(\overline{d}), \psi_{asct}, Y_{asct}, s_0 s_1 \dots s'_i \dots s'_j s_n s'_m)$;
else
$t_{asct} = (s_0 s_1 \dots s_i \dots s_j \dots s_n s_m, \alpha(\overline{d}), \psi_i, \psi_j, Y_i, Y_j, s_0 s_1 \dots s'_i \dots s'_j s_n s_m)$
if $clock\_order(t_{asct})$ **then**
$T_{asct} = T_{asct} \cup t_{asct}$;
$\overline{s} = s_0 s_1 \dots s'_i \dots s'_j s_n s'_m$, return true;
else
$t_{med} = mediateur(t_i, D)$;
if $t_{med} \neq null$ **then**
$t_{asct} = (s_0 s_1 \dots s_i \dots s_n s_m, \alpha(\overline{d}), \psi_i, Y_i, s_0 s_1 \dots s'_i \dots s_n s'_m)$ **if**
$clock\_order(t_{asct})$ **then**
$T_{asct} = T_{asct} \cup t_{asct}$;
$\overline{s} = s_0 s_1 \dots s'_i \dots s_n s'_m$, return true;
else
return false;
else
return false;
else
$t_{med} = mediateur(t_i, D)$;
if $t_{med} \neq null$ **then**
$t_{asct} = (s_0 s_1 \dots s_i \dots s_n s_m, \alpha(\overline{d}), \psi_i, Y_i, s_0 s_1 \dots s'_i \dots s_n s'_m)$;
if $clock\_order(t_{asct})$ **then**
$T_{asct} = T_{asct} \cup t_{asct}$;
$s_{asct} = s_0 s_1 \dots s'_i \dots s_n s'_m$, return true;
else
return false;
else
15 return false;

The algorithm 4.3 allows to define an order between the different clocks of services. Based on the computed order, it detects timed conflicts. This algorithm has as input a candidate TCSA transition $t_i = (s_i, m_i(\overline{d}), c_i, \psi_i, Y_i, s'_i)$. To discover timed conflicts, it proceeds as follows.

- It propagates timed constraints, of the form $x > v$ (resp. $x \geq v$), from a predecessor transition

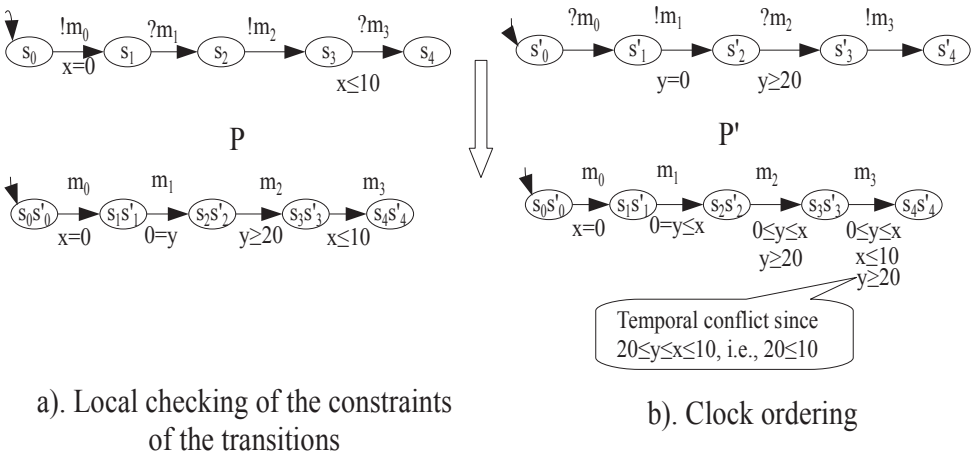


Figure. 4. Make explicit the implicit timed constraints dependency.

t_{i-1} to the transition t_i.

- A clock z which is reset in a predecessor transition t_{i-1}, has a value bigger than a clock y which is reset in the current transition t_i. Hence, it defines the order $y \leq z$.

- In addition, it propagates the order $z_1 \leq \ldots \leq z_n$ of the predecessor transition t_{i-1}.

- If in the transition t_i there exists a constraint of the form $x \leq v$ (resp. $x \geq v$) and at the same time, a clock y is reset, then it defines the order $x - y \leq v$ (resp. $x - y \geq v$). That means, the difference between the two clocks x and y is always less (resp. bigger) than v.

- If among the set of constraints and defined orders, there exists two constraints $x \geq v$ and $x' \geq v'$, and at the same time, there is an order of the form $x - x' \geq v$, it implies the order $x \geq v + v'$. In fact, this order allows to consider the clocks value accumulation.

By applying theses steps when building TCSA transitions, timed conflicts are discovered if at least one of the following conditions is satisfied.

- There exists an order of the form $v \leq x_1 \leq \ldots \leq x_n \leq v'$ where $v' \leq v$.

- There exists three constraints $x \geq v'$ and $y \leq v''$ and $x - y \leq v$ with $v' - v'' > v$ (i.e., following the constraints $x \geq v'$ and $y \geq v''$, the difference $x - y \leq v$ is violated).

- There exists three constraints $x \leq v'$, $y \leq v''$ and $x - y \geq v$ with $v' < v$ (i.e., the constraint $x - y \geq v$ is violated),

- There exists three constraints $x \leq v'$, $y \geq v''$, and $x - y \geq v$ with $v' - v'' < v$ (i.e., the constraint $x - y \geq v$ is violated)

4.3 Generation of a timed mediator

As said previously, because of timed (and non-timed) conflicts, timed P2P connection process can fail. The mediator aims to resolve these conflicts by creating required messages. In our approach, a required message is created taking the involved data from the history of past exchanged messages, i.e., the current available data (we assume here that data having the same name, have also the same value).

In order to produce the required messages, we check if the involved data are available, i.e, they have been already exchanged. In other terms, the mediator reuses the data historic to produce the required messages.

The mediator is defined using the computed TCSA, by adding input, output and empty messages.

Algorithm 4.3: Clock_Order

Input: A transition $(s_i, m_i(\overline{d}), \psi_i, Y_i, s'_i)$
Output: boolean
begin

1 **if** s_i *is the initial state* **then**
2 return true;
else
for *each* $\varrho_{i-1} \in \psi_{i-1}$, *such as* $\varrho_{i-1} = x \geq v$ *or* $\varrho_{i-1} = x > v$ *of* $(s_{i-1}, m_{i-1}(\overline{d}), \psi_{i-1}, Y_{i-1}, s'_{i-1})$ **do**
3 $\psi_i = \psi_i \cup \varrho_{i-1}$;
for *each* $y = 0 \in Y_i$ *and* $z = 0 \in Y_{i-1}$ **do**
4 $\psi_i = \psi_i \cup y \leq z$;
for *each* $z_1 \leq z_2 \in \psi_{i-1}$ **do**
5 $\psi_i = \psi_i \cup z_1 \leq z_2$;
for *each* $y \in Y_i$ *et* $\varrho_i \in \psi_i$ **do**
6 **if** $\varrho_i = x \leq v$ **then**
$\psi' = x - y \leq v$;
$\psi_i = \psi_i \cup \psi'$
else
if $\varrho_i = x \geq v$ **then**
7 $\psi' = x - y \geq v$;
$\psi_i = \psi_i \cup \psi'$
8 **if** $\exists \varrho_i = x \geq v$ *and* $\varrho'_i = x' \geq v'$ *and* $\varrho''_i = x - x' \geq v$ **then**
9 $\psi' = x \geq v + v'$ $\psi = \psi \cup \psi'$
10 **if** $\exists\ v \leq y_0 \leq \ldots \leq y_n \leq v' \in \psi_i$ *such as* $v' < v$ **then**
return false;
else
11 **if** $\exists\ x \geq v' \in \psi_i$ *and* $y \leq v'' \in \psi_i$ *and* $x - y \leq v \in \psi_i$ *such as* $v' - v'' > v$ **then**
return false;
else
12 **if** $\exists\ x \leq v' \in \psi_i$ *and* $y \leq v'' \in \psi_i$ *and* $x - y \geq v \in \psi_i$ *such as* $v' \geq v''$ *and* $v' < v$ **then**
return false;
else
13 **if** $\exists\ x \leq v' \in \psi_i$ *and* $y \geq v'' \in \psi_i$ *and* $x - y \geq v \in \psi_i$ *such as* $v' - v'' < v$ **then**
return false;
else
14 return true;

As long as the TCSA can be executed, the mediator does nothing. When two services can exchange a message and there are clocks which are reset, the mediator resets the same clocks via an empty transition. In fact, these clocks can be used later by the mediator to consume messages within a defined time window. Whilst, when there is a deadlock, the mediator generates the required message to avoid the conflict, if possible. The following steps are used to generate a mediator:

(1) If the mediator must be generated to consume an extra message produced by a transition $(s_i, !m(\overline{d}), c_i, \psi_i, Y_i, s'_i)$, then

(2) We generate the mediator transition $(s_m, ?m(\bar{d}), c_i, \psi_i, Y_i, s'_m)$
(3) If the mediator must be generated to produce a required message, then
(4) If the involved data are available, then
(5) The mediator generates the missing message,
(6) Else, if at least there is one missing data, then the mediator cannot produce the missing message, and
(7) Return null

Algorithm 4.4: Generation of a mediator

Input: A transition $t_i = (s_i, \alpha(\bar{d}), c, \psi, Y, s'_i)$, the set of exchanged data D_a
Output: A transition
mediator$((s_i, \alpha_i, c_i, \psi_i, Y_i, s'_i), D_a, \bar{s})$;
begin
1 **if** $\alpha = !m(\overline{d_m})$ **then**
  $M_{med} := M_{med} \cup m(\bar{d})$;
  $S_{med} := S_{med} \cup s'_m$;
2 $T_{med} := T_{med} \cup (s_m, ?m(\bar{d}), c_i, \psi_i, Y_i, s'_m)$;
  return $(s_m, ?m(\bar{d}), c_i, \psi_i, Y_i, s'_m)$;
else
3 **if** $\alpha = ?m(\overline{d_m})$ **then**
4 **if** $\overline{d_m} \subseteq D_a$ **then**
  $M_{med} := M_{med} \cup m(\overline{d_m})$;
  $S_{med} := S_{med} \cup s'_m$;
5 $T_{med} := T_{med} \cup (s_m, !m(\overline{d_m}), c_i, Y_i, s'_m)$;
  return $(s_m, ?m(\bar{d}), c_i, Y_i, s'_m)$;
  else
6 The required data are not available, so the required message cannot be produced;
7 return null;

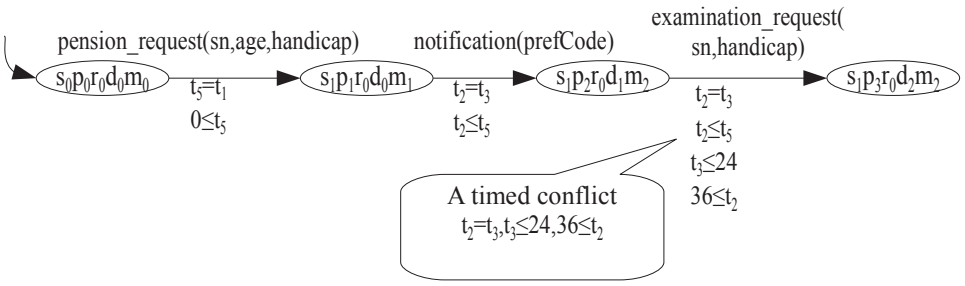
5. BACK TO THE CASE STUDY

In order to illustrate the approach presented in this paper, we propose to show a concrete composition example using the *PS*, *HAS*, *THS* services, and the client service introduced in Section 2. We first try to build a TCSA (Fig. 5.a) without the timed involvement of a mediator, i.e., the mediator will be involved only to resolve non-timed conflicts (Fig 5.b).

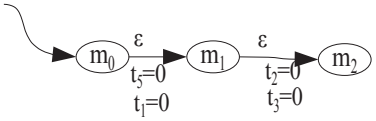
5.1 Composition without the timed involvement of the mediator

As in our framework, a mediator can be involved, we generate an empty mediator that has initially only one state m_0. The initial configuration of the *TCSA* is $s_0p_0r_0d_0m_0$ (respectively the client service, PS, HAS, THS, and the mediator are in their initial states). From the current state of the client service s_0, the message $!pension\_request(sn, age, handicap)$ can be sent. As we can remark, the *PS* service waits for this message. Since, the constraints over data ($age \geq 18$ and $age \geq 16$) are not disjoint, we can connect the two transitions $(s_0, !pension\_request(sn, age, handicap), age \geq 18, t_5 = 0, s_1)$ and $(p_0, ?pension\_request\ (sn, age, handicap), age \geq 16, t_1 = 0, p_1)$. When the two transitions are fired, the two clocks t_1 and t_5 are reset. So, we generate an empty mediator transition that allows to reset the same clocks. In fact, theses clocks can be used later to

specify constraints to produce or consume messages. We build a global TCSA transition that connects the two transitions of the client and *PS* services with the transition of the mediator $(s_0p_0r_0d_0m_0, pension\_request(sn, age, handicap), t_1 = t_5 = 0, s_1p_1r_0d_0m_1)$. The new configuration becomes $s_1p_1r_0d_0m_1$ and the new current state of the client service becomes s_1. From this new configuration, the current transition of the client service is $(s_1, ?medical\_examination(reason), s_2)$. There is no transition that enables sending the message $medical\_examination(reason)$. So we check if the mediator can produce this message. Since the data *reason* has not been already exchanged, the mediator cannot generate the message $medical\_examination(reason)$. Among the services transitions, we choose the transition $(p_1, !notification(prefCode), t_2 = 0, p_2)$. Since, the *HAS* service can consume it, we can connect them. As the clocks t_2 and t_3 are reset, we generate an empty mediator transition that reset the same clocks. We build the TCSA transition $(s_1p_1r_0d_0m_1, notification(prefCode), t_2 \leq t_5, t_2 = t_3, s_1p_2r_0d_1m_2)$.



a)- A conflicted TCSA



b)- The associated mediator

Figure. 5. Composition without the timed involvement of the mediator

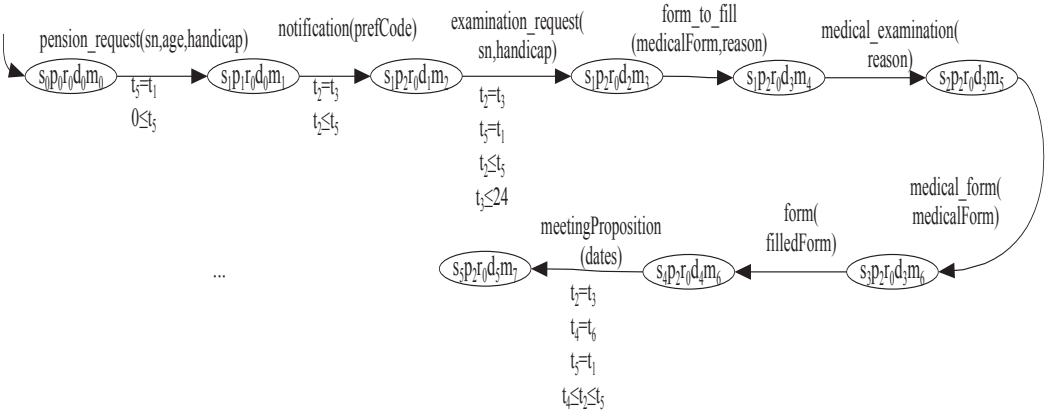
From the new configuration, the HAS service waits for the message $examination\_request(sn, handicap)$ that must be consumed within 24 hours from receiving the message $notification(codePref)$. The message $examination\_request(sn, handicap)$ can be sent by the *PS* after 36 hours from sending the message $notification(codePref)$. We build the TCSA transition $(s_1p_2r_0d_1m_2, examination\_request(sn, handicap), t_2 = t_3, t_2 \leq t_5, t_3 \leq 24, t_2 \geq 36, s_1p_3r_0d_2m_2)$. This transition is conflicting, since $t_2 = t_3$, $t_3 \leq 24$ et $t_2 \geq 36$. Thus, we can see that without involving the mediator to handle timed conflicts, the compositions fails.

5.2 Involving the mediator

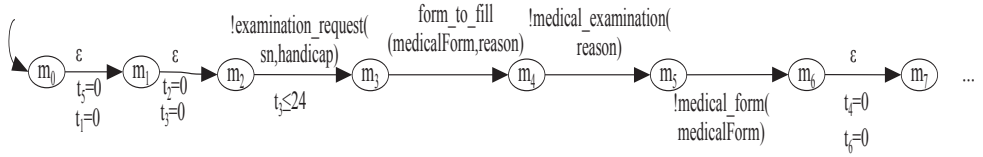
We show here how the mediator can be involved to handle timed conflicts.

To generate the TCSA (Fig. 6.a) and the associated timed mediator (Fig. 6.b), we use the following steps. We apply the same steps described above to reach the configuration $s_1p_2r_0d_1m_2$. From this configuration, the HAS service can fire the passive transition $(d_1, ?examination\_request(sn,$

handicap), $t_3 \leq 24, d_2$). Since the corresponding transition of the PS service (p_2, !*examination_request*(*sn*, *handicap*), $t_2 \geq 36, p_3$) is a non-passive transition, we check if the mediator can generate the message *examination_request*(*sn*, *handicap*). The data *sn*, and *handicap* have been already exchanged. Hence, the mediator can generate the required message *examination_request*(*sn*, *handicap*) via the transitions (m_2, !*examination_request*(*sn*, *handicap*), $t_3 \leq 24, m_3$). When the message is generated, we build the global transition ($s_1p_2r_0d_1m_2$, *examination_request*(*sn*, *handicap*), $t_2 = t_3, t_2 \leq t_5, t_3 \leq 24, s_1p_2r_0\ d_2m_3$). From the new configuration $s_1p_2r_0d_2m_3$, we choose the passive transition (d_2, !$form\_to\_fill$(*medicalForm*, *reason*), d_3) of the HAS service. As there is no service that waits for the message $form\_to\_fill$(*medicalForm*, *reason*), we generate the mediator transition to consume this message, i.e., (m_3, ?$form\_to\_fill$(*medicalForm*, *reason*), m_4), and then we build the global transition ($s_1p_2r_0d_2m_3$, $form\_to\_fill$(*medicalForm*, *reason*), $s_1p_2r_0d_3m_4$).



a)- A part of the generated TCSA



b)- A part of the associated mediator

Figure. 6. The timed composition schema automaton (TCSA).

From the new configuration, the current client transition is (s_1, ?*medical_examination*(*reason*), s_2). There is no transition that enables sending the message *medical_examination*(*reason*). The mediator can produce the message *medical_examination*(*reason*), via the transition (m_4, !*medical_examination*(*reason*), m_5), and then we build the TCSA transition ($s_1p_2r_0d_3m_4$, *medical_ examination*(*reason*), $s_2p_2r_0d_3m_5$). The current transition of the client service is (s_2, ?*medical_form*(*medicalForm*), s_3). The data *medicalForm* has been already sent by the HAS service. So, the mediator can generate the

missing message *medical_ form*(*medicalForm*) via the transition (m_5, !*medical_form*(*medicalForm*), m_6). Once the transition of the mediator is generated, we build the global transition ($s_2p_2r_0d_3m_5$, *medical_form*(*medicalForm*), $s_3p_2r_0d_3m_6$). From the new configuration, we connect respectively the two transitions of the client and *HAS* services (s_3, !$form(filledForm), s_4$) and (d_3, !$form(filled-Form), d_4$) via the TCSA transition ($s_3p_2r_0d_3m_6$, $form($ $filledForm), s_4p_2r_0d_4m_6$).

By applying the same steps, either we build the TCSA, or we detect a conflict that cannot be avoided.

6. FORMAL VERIFICATION AND VALIDATION OF THE BUILT COMPOSITION

As presented previously, when the composition succeeds, the algorithm generates a mediator and produces a global timed composition schema TCSA. Such a built TCSA is an *optimized product* built on the fly: indeed, we build progressively the product of timed protocols rather than building the whole product.

The built TCSA is correct if it is deadlock free and it satisfies the client service. Checking that the TCSA is deadlock free can be reduced to checking reachability properties. This problem is PSPACE-complete in general. The problem of client service satisfaction checking can be reduced to the *inclusion problem*, which is decidable [1]. In fact, the formal model of timed conversational protocol that we have defined relies on a deterministic timed automata for which closure and decidability properties have been proved [1].

In the following, we present a formal verification process which aims to validate the built composition. We note that this verification process is generic and can be used to verify atomic and composite services built automatically or manually. This process relies on a model checking approach inspired from [14] and using the UPPAAL model checker. However, despite success improvements in UPPAAL's performance, the state explosion problem is a still a reality [2].

6.1 UPPAAL overview

UPPAAL is a model checker for the verification and simulation of real time systems [22]. An UPPAAL model is a set of timed automata, clocks, channels for systems (automata) synchronization, variables and additional elements [22].

Each automaton has one initial state. Synchronization between different processes can take place using channels. A channel can be written into (denoted as *channel_name*!), and can be read (denoted as *channel_name*?). A channel can be defined as *urgent* to specify that the corresponding transition must be fired as soon as possible, i.e. immediately and without a delay. Variables and clocks can be associated to processes (automaton). Conditions on these clocks and variables can be associated to transitions and states of the process. The conditions associated to transitions, called *guards*, specify that a transition can be fired if the corresponding guards are satisfiable. The conditions associated to states, called *invariants*, specify that the system can stay in the state while the invariant is satisfiable.

The UPPAAL properties query language is a subset of *Computation Tree Logic* (CTL) [19]. The properties that can be analyzed by UPPAAL are:

$A[]\psi$: for all the automata' paths, the property ψ is always satisfiable, i.e., for each transition (or a state) of each path, the property ψ is satisfiable.

$A <> \psi$: for all the automata' paths, the property ψ is eventually satisfiable, i.e., for each path , there is at least one transition (or a state) in which the property ψ is satisfiable.

$E[]\psi$: there is at least a path in the automata such that the property ψ is always satisfiable, i.e., there is at least one path such that for each transition (or a state), the property ψ is satisfiable.

$E <> \psi$: there is at least a path in the automata such that the property ψ is eventually satisfiable, i.e., there is at least one transition (or a state) of at least one path in which the property ψ is satisfiable.

$\psi \rightsquigarrow \phi$: when ψ holds, ϕ must hold.

In the following, we present the formal primitives we propose for composition checking.

6.2 Verification of Web service compositions

In this section, we present the verification process we propose using the model checker UPPAAL. The purpose of this verification process is to check if the built composition holds deadlocks. In this context, we define three composition classes: (1) *fully correct composition* , (2) *partially correct composition*, (3) *incorrect composition*.

6.2.1 *Fully correct composition.* We say that a composition is correct if it is (timed and non-timed) deadlock free. This is equivalent to check that its corresponding TCSA does not hold timed and non-timed conflicts.

Formally, checking that a composition is fully correct is equivalent to check that all the paths of the TCSA lead to a final state.

Let Q be a TCSA and s_f its final state. Q is said to be fully correct, iff the following CTL formula is correct:

$$\boxed{A <> \ Q.s_f} \tag{1}$$

6.2.2 *Partially correct composition.* A composition is said to be incorrect if its TCSA is not deadlock free. Formally, a composition is not fully correct if there exists at least a path of the TCSA which does not lead to a final state. This latter can be specified as the following CTL formula:

$$\boxed{E[] \ not \ Q.s_f} \tag{2}$$

When the composition is not fully correct, we check if it can achieve at least one correct execution. Formally, a composition can terminate at least one execution if its final state can be reached via at least one path. The former property can be specified as follows.

$$\boxed{E <> \ Q.s_f} \tag{3}$$

A composition is said to be partially correct if it is not fully correct (i.e., the property 2 is satisfiable) but at the same time it can fulfil at least one execution (i.e., the property 3 is satisfiable).

6.2.3 *Incorrect composition.* When the composition is not even partially correct, we say that the composition is fully incorrect. As specified by the following CTL formula, a composition is said to be fully incorrect if all its TCSA paths do not lead to a final state.

$$\boxed{A[] \ not \ Q.s_f} \tag{4}$$

7. EVALUATION AND DISCUSSION

In this section, we first study the properties of the algorithm we propose. In the second part, we expose the experimentation setup of the proposed framework.

7.1 Properties of the composition algorithm

We discuss the algorithm terminating, soundness, and completeness.

The composition algorithm terminates: Suppose that the composition algorithm does not terminate. In that case, there is at least one trace of the TCSA that has infinite transitions. This latter happens if the services hold states that can be visited infinite times (i.e., there is a loop). Our algorithm caters loops. As we have a finite number of states, and each loop is parsed a finite number of times, our algorithm does not loop. As we deal with finite number of transitions, our algorithm either constructs a TCSA transition when two services can be connected, or fails. Thus, it terminates.

The composition algorithm is sound: Suppose that the composition algorithm is not sound. In other terms, the algorithm either builds a TCSA in spite that the services cannot be connected, or the algorithm fails to connect services which can be connected.
In the first case, the algorithm builds a TCSA. According to our work, a TCSA transition $(\overline{s}, m(\overline{d}), c, \psi, Y, \overline{s'})$ that characterizes that the message $m(\overline{d})$ goes through a channel (i.e., there is a service that sends the message $m(\overline{d})$ and there is another service that receives it so that there is no a timed conflict that arises). So, the algorithm generates a TCSA transition only when two services can be connected.
The second case concerns the fact that there is a transition which cannot be connected. This happens if there is no a counterpart transition, or there is a timed conflict. So, our algorithm does not build an accepted TCSA transition when it is not possible.

The composition algorithm is complete: Now we discuss the completeness of the composition algorithm. Suppose that the algorithm fails to build a composition. This happens if: (1) there is at least one transition of the client service that cannot be satisfied, or (2) the client service can be satisfied, but there is at least one service that does not reach its final state.
The algorithm we propose checks if each transition of the client service can be connected. If it fails, the algorithm checks if there is a transition of the available services that can be fired. If no transition can be fired, the algorithm returns false and the composition fails.
Now we tackle the second case in which the client service is satisfied (all the transitions of the client service are fired) but there is at least one transition of the available services that cannot reach its final state. The algorithm checks when it terminates, if all the transitions of the client service are satisfied (i.e., are all fired). In that case, it checks if all the available services reach their final states. If there is at least one transition of the services that cannot reach its final state, thus, the composition fails.

Suppose that the algorithm succeeds to build a composition. We show that the algorithm looks for all the possibilities. A composition succeeds if all the transitions of the client service can be fired and all the involved services reach their final states. According to the algorithm 4.1, all the transitions of the client service are checked. Once the algorithm terminates, we check if all the transitions of the client service are satisfied and then we check if all the services can reach their final states.

7.2 Experimentation

In order to experiment the proposed approach, a prototype has been implemented [13]. Its underlying architecture is depicted in Fig. 7. The tool inputs the description of services and the client service as XML documents. The *P2P coordination* component tries to build P2P connections (channels) among the services and updates the TCSA description thanks to the algorithm 4.1. A third component, the *timed mediator* component, steps in to consume extras messages or to produce, if possible, the required messages using the algorithm 4.4. We note that the data historic repository is a database in which we store the involved exchanged data.

To evaluate the proposed approach, we conducted preliminary experiments using the prototype we implemented. The experiments have been done on a laptop, 1,66 GHz CPU, and 1 Go

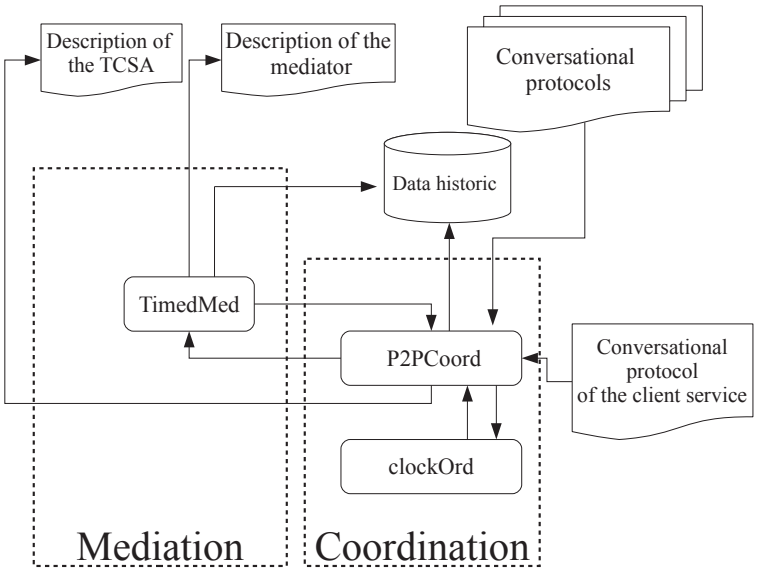


Figure. 7. Underlying architecture of the prototype.

of memory. We have obviously remarked that the performance of the approach we propose is proportional to the complexity of the client and involved services.

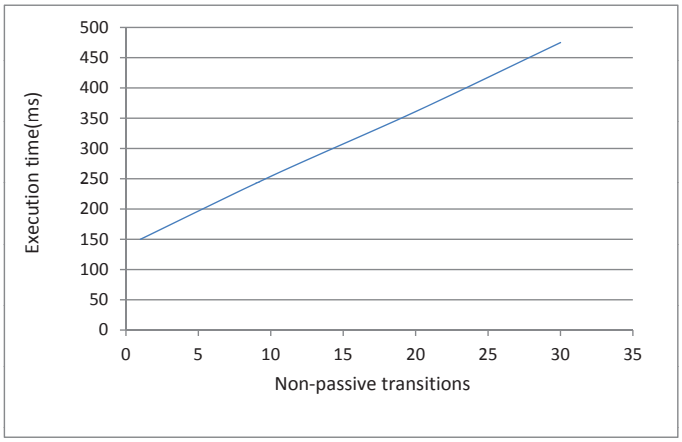


Figure. 8. Performance evaluation of the proposed approach according to the non-passive transitions.

Moreover, the performance of the proposed approach is more sensible to the number of client transitions that cannot be satisfied by services. In fact, for each client transition, the algorithm checks the services to find the one that can satisfy the transition. In case the transition cannot be satisfied, the algorithm checks all the services to try to connect them. So, more the number of non-satisfied client transitions increases and the Web services are complex, more the complexity of the algorithm increases too.

In addition, we have compared the impact of non-passive and passive transitions. Fig 8 shows the execution time (in millisecond) evaluation with respect to the number of non-passive transitions. While non-passive transitions affect considerably the performance of the composition algorithm, as seen in Fig 9, the impact of passive transitions is lesser. Indeed, when a passive transition is reached, the algorithm tries to connect it with the counterpart transition. Whilst,

when a non-passive transition is reached, before trying to connect it, the algorithm tries first to connect all potential passive transitions. When all passive transitions are checked, the algorithm tries to connect the non-passive transition.

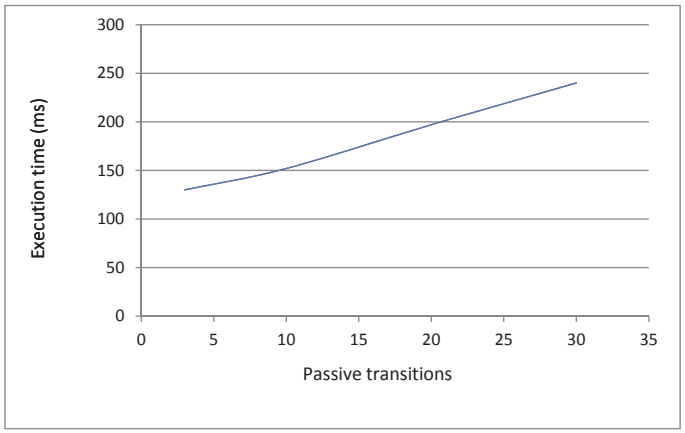


Figure. 9. Performance evaluation of the proposed approach according to the passive transitions.

8. RELATED WORK AND DISCUSSION

The research field about how to synthesize automatically a composition is very active. Several research works have been published on automatic service composition, using techniques based on situation calculus [29; 34], transition based systems [10; 28; 12], or symbolic model-checking applied to planning [31]. Unlike the proposed approaches, in our framework we cater for timed properties when composing services.

In [11] the authors consider a service as a data-driven entity characterized by a database and a set of Web pages. When a Web page is activated, a set of input choices are presented to the client. The client chooses one of such inputs, and in response, the service produces as output updates over the service database and/or performs some actions, and makes a state transition, which is seen by the client as a transition from a web page to another. A service is therefore seen as a tree of web pages. There, Web services are atomic and authors do not consider the control flow which is an important aspect in the Web service composition. Moreover, there is no timed considerations as in our work.

In [35; 36; 37] the authors consider services as views over data sources. They build on the idea that heterogeneity of data sources may be overcome by exploiting services as wrappers of different information sources, thus providing uniform access to them, exploiting standard protocols such as SOAP and XML. Each data source, i.e., service, is described in terms of input and output parameters (the latter provided by the source), binding patterns and additional constraints on the source. The latter allow to characterize the output data. Analogously, these works consider only atomic services. However, the control flow between data is a crucial aspect. Furthermore, the authors do not consider timed properties.

Like the above works in [26; 23], the considered Web services are atomic. The behavioral aspect is not considered and the timed aspects are not taken into account.

In [32; 8], Web services are described by their BPEL specification. The authors proposed to translate the BPEL specifications into a finite state machine (FSM) specification. As these composition approaches are not oriented by the client need (there is no client need notion), the composition consists in performing the product of the whole FSM specification. The composition problem consists then to find paths in the computed cartesian product that satisfies reachability properties. According to our work, the works presented in [32; 8] do not cater for timed

properties when building a composition. Moreover, in [32; 8], the authors do not deal with the problem of missing messages, since they do not consider data and communications capabilities as in our framework. In addition, our composition approach is oriented by the client need, defined upon the required data flow, that allows to optimize the cartesian product: we compose only the relevant parts of the services and not the whole services as in [32; 8].

In [10], Web services exchange asynchronous messages and they are modelled as Mealy machines. The authors investigate an approach dealing with the unexpected interactions between the local and global behavior of composite Web services. However, only messages without parameters are considered. Moreover, the authors are not concerned with how composing services but they are interested in analyzing the local and global behavior of Web services in a composition. Furthermore, the authors do not deal with timed properties.

An other remark is that, works that consider the control flow, address the composition problem at process level, i.e., they consider the operations the services perform [20; 5; 7; 9; 32]. For example, in [6; 5], one of the important assumptions is that the client need (called *goal service*) is specified upon the operations of the services. The precise specification of the goal service allows for precise matching with available, more elementary services. Nevertheless, in real life scenarios, it is not always possible for a client to precisely specify his need according to the operations of the services. A simple client does not have any preliminary knowledge about the service's operations. Whilst, in our framework, the client need (client service) is specified by the (input and output) data the client expects. Moreover, in [6; 5], the authors do not deal with timed properties when composing services.

The few frameworks that deal with timed properties in Web services specification, focus on compatibility and replaceability analysis [33] and timed model checking a given composition [21]. In both works, the authors consider synchronous Web services. While, in our work, we deal with asynchronous Web services. If services can be synchronized, then we build P2P connections. Else, when the services cannot be synchronized directly, we generate the mediator that tries to interconnect indirectly the asynchronous services.

These works do not deal with how to synthesize a composition of asynchronous services by considering their timed properties. When building a timed composition, we cater for the problem of missing messages. However, in [21], the authors do not deal with how building a composition, but their assumption is that the composition is already built.

9. CONCLUSION AND PERSPECTIVES

In this paper, we presented a formal approach to handle timed properties in asynchronous Web services composition. Our framework is oriented by the client data flow. We first proposed a timed automata based formal model of timed conversational protocols. This model provides an operational semantic to consider timed properties of asynchronous communicating Web services. This model gathers: (1) *supported messages*, (2) *data*, (3) *constraints over data*, (4) *timed constraints*, and (5) the *asynchronous conversational aspect* of Web services. Based on the model we proposed, we provided an algorithm which aims at building a composition so that no timed conflicts arise. In this context, we use *the clock ordering process* that allows to discover implicit timed conflicts that can arise when composing services.

Unfortunately, due to the heterogeneous nature of Web services, timed P2P connections can fail, and the composition too. To tackle this problem, we proposed to generate a third party service, called *mediator*. The role of this latter is to avoid conflicts. Obviously, the mediator has a crucial role when composing services, since it contributes to connect the required services by producing the expected messages.

The proposed approach has been implemented in a prototype, which has been used to perform preliminary experiments. Currently, we are trying to carry out fine grained experimentations on a set of richer services.

The framework we presented in this paper focuses on the composition of timed asynchronous services and considers correct interactions of services. Our ongoing work studies the problem of exceptions handling within the timed composition framework. Moreover, we plan to extend our approach with semantic capabilities in order to support more complex timed properties. This will allow us to construct a composition not only by considering timed properties associated to exchanging messages, but also more global constraints.

Another interesting research direction consists in studying dynamic instantiation related problems when composing timed Web services. In this paper, we assume that only one instance of each service is required. However, in real scenarios, we can need one or several instances of each service. We believe that it is interesting to extend the proposed approach to handle such features.

As our framework is mainly oriented by data capabilities, another important aspect to consider is data privacy issues of Web services. This will allow to get control over the use of data.

REFERENCES

R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi. Uppaal implementation secrets. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 3–22, 2002.

B. Benatallah, F. Casati, J. Ponge, and F. Toumani. On temporal abstractions of web service protocols. In *The 17th Conference on Advanced Information Systems Engineering (CAiSE '05). Short Paper Proceedings*, 2005.

B. Benatallah, F. Casati, J. Ponge, and F. Toumani. Compatibility and replaceability analysis for timed web service protocols. In *Proceedings of the 21èmes Journées Bases de Données Avancées (BDA'05)*, Saint Malo, 17-20 octobre 2005,.

D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 613–624. ACM, 2005.

D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Service-Oriented Computing - ICSOC 2003, First International Conference, Trento, Italy, December 15-18, 2003, Proceedings*, volume 2910 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2003.

D. Berardi, D. Calvanese, G. D. Giacomo, and M. Mecella. Composition of services with nondeterministic observable behavior. In *Service-Oriented Computing - ICSOC 2005, Third International Conference (ICSOC)*, pages 520–526, 2005.

P. Bertoli, M. Pistore, and P. Traverso. Automated web service composition by on-the-fly belief space search. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006 (ICAPS)*, pages 358–361, 2006.

A. Brogi and R. Popescu. Towards semi-automated workflow-based aggregation of web services. In *Service-Oriented Computing - ICSOC 2005, Third International Conference (ICSOC)*, pages 214–227, 2005.

T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *Proceedings of the international conference on World Wide Web, WWW 2003*, pages 403–410, 2003.

A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web services. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'04)*, pages 71–82, June 14-16, Paris, France, 2004.

G. Díaz, M.-E. Cambronero, J. J. Pardo, V. Valero, and F. Cuartero. Automatic generation of correct web services choreographies and orchestrations with model checking techniques. In *Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW'06)*, page 186, 19-25 February, Guadeloupe, French Caribbean, 2006.

N. Guermouche. *Timed Interation-aware Web service composition (Wrtiten in French: Etude des Interactions Temporises dans la Composition de Services Web)*. PhD thesis, Nancy university, France, 2010.

N. Guermouche and C. Godart. Timed model checking based approach for web services analysis. In *IEEE International Conference on Web Services (ICWS'09), July 6-10, 2009, Los Angeles, CA, USA*, 2009.

N. Guermouche and C. Godart. Timed properties-aware asynchronous web service composition. *Proceedings of the* 16^{th} *International Conference on COOPERATIVE INFORMATION SYSTEMS (CoopIS'08)*, pages 44–61, Monterrey, Mexico, November 9-14, 2008.

N. Guermouche and C. Godart. Toward data flow oriented services composition. In *Proceedings of the* 12^{th} *International IEEE Enterprise Distributed Object Computing Conference (EDOC'08)*, pages 379–385, Munich, Germany, September 15-19, 2008.

N. Guermouche and C. Godart. Timed conversational protocol based approach for web services analysis. In *Proceedings of the* 8^{th} *International Conference on Service Oriented Computing (ICSOC'10)*, San Francisco, California, USA, December 7-10, 2010.

R. Hamadi, H.-Y. Paik, and B. Benatallah. Conceptual modeling of privacy-aware web service protocols. In *Proceedings of the* 19^{th} *International Conference on Advanced Information Systems Engineering (CAiSE'07)*, pages 233–248, Trondheim, Norway, June 11-15, 2007.

T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: a look behind the curtain. In *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–14, 2003.

R. Kazhamiakin, P. K. Pandya, and M. Pistore. Representation, verification, and computation of timed properties in web service compositions. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pages 497–504, 2006.

K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. In *International Journal on Software Tools for Technology Transfer*, 1997.

S. A. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *Proceedings of the* 8^{th} *International Conference on Principles and Knowledge Representation and Reasoning (KR'02)*, pages 482–496, April 22-25, Toulouse, France,2002.

M. Mecella and C. Batini. Enabling italian e-government through a cooperative architecture. *IEEE Computer*, 34(2):40–45, 2001.

M. Mecella, M. Ouzzani, F. Paci, and E. Bertino. Access control enforcement for conversation-based web services. In *WWW 2006 Proceedings*, 2006.

B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *VLDB J.*, 12:333–351, 2003.

S. B. Mokhtar, A. Kaul, N. Georgantas, and V. Issarny. Efficient semantic service discovery in pervasive computing environments. In *7th International Middleware Conference, (Middleware)*, pages 240–259, Melbourne, Australia, November 27-December 1, 2006.

A. Muscholl and I. Walukiewicz. A lower bound on web services composition. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, volume 4423 of *LNCS*, pages 274–287, 2007.

S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the international conference on World Wide Web, WWW 2002*, pages 77–88, 2002.

H. Pichler, M. Wenger, and J. Eder. Composing time-aware web service orchestrations. In *Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE'09)*, pages 349–363, Amsterdam, The Netherlands, June 8-12, 2009.

M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated composition of web services by planning at the knowledge level. In *IJCAI*, pages 1252–1259, 2005.

M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite bpel4ws web services. In *IEEE International Conference on Web Services (ICWS)*, pages 293–301, 2005.

J. Ponge, B. Benatallah, F. Casati, and F. Toumani. Fine-grained compatibility and replaceability analysis of timed web service protocols. In *the 26th International Conference on Conceptual Modeling (ER)*, 2007.

S. Sohrabi, N. Prokoshyna, and S. A. McIlraith. Web service composition via generic procedures and customizing user preferences. In *International Semantic Web Conference*, pages 597–611, 2006.

S. Thakkar, J. L. Ambite, and C. A. Knoblock. A view integration approach to dynamic composition of web services. In *Proceeding of 2003 ICAPS Workshop on Planning for Web Services*, 2003.

S. Thakkar, J. L. Ambite, and C. A. Knoblock. A data integration approach to automatically composing and optimizing web services. *Proceedings of the 2nd ICAPS International Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

S. Thakkar, J. L. Ambite, C. A. Knoblock, and C. Shahabi. Dynamically composing web services from on-line sources. *Proceeding of the AAAI Workshop on Intelligent Service Integration*, pages 1–7, 2002.